Modeling self-adaptative IoT architectures

Iván Alfonso* †, Kelly Garcés*, Harold Castro* and Jordi Cabot† ‡
*Department of Systems and Computing Engineering
Universidad de los Andes, Bogotá, Colombia
Email: {id.alfonso, kj.garces971, hcastro}@uniandes.edu.co
†Universitat Oberta de Catalunya, Barcelona, Spain
‡ICREA, Barcelona, Spain
Email: jordi.cabot@icrea.cat

***Abstract*—An Internet of Things (IoT) architecture describes a network of (physical) objects exchanging data over the Internet. In complex IoT deployments, as those typical of Industry 4.0 systems, this architecture comprises four different layers: device, edge, fog, and cloud. Moreover, the composition and computation tasks assigned to each layer may dynamically evolve due to environmental changes.**

As such, modeling IoT architectures is a complex process that must cover as well the specification of self-adaptation rules to ensure the optimized execution of the IoT system. To facilitate this task, we propose a new IoT Domain-Specific Language (DSL) covering both the static and dynamic aspects of an IoT deployment. The definition of the DSL, its projectional-based editor and a first prototype of a Kubernetes manifest generator are available as open source software.

***Index Terms*—Domain-specific language, Internet of Things, Self-adaptive system, Edge computing, Fog computing**

I. Introduction

The ideas behind the IoT have been especially embraced by industry in the so-called Industrial IoT (IIoT) or Industry 4.0. Currently billions of devices become connected with potential capabilities to sense, communicate, and share information about their environment. Traditional IoT systems rely on cloud-based architectures, which allocate all processing and storage capabilities to cloud servers. Although cloud-based IoT architectures have advantages such as reduced maintenance costs and application development efforts, they also have limitations in bandwidth and communication delays [1]. Given these limitations, edge and fog computing have emerged with the goal of distributing processing and storage close to data sources (i.e. things). Today, developers tend to leverage the advantages of edge, fog, and cloud computing to design multi-layered architectures for IoT system.

Nevertheless, creating such complex designs is a challenging task. Even more challenging is managing, and adapting IoT systems at runtime to ensure the optimal performance of the system while facing changes in the environment conditions. IoT systems are commonly exposed to changing environments that induce unexpected events at run-time (such as unstable signal strength, latency growth, and software failures) that can impact its Quality of Service (QoS). To deal with such events, a number of runtime strategies (such as auto-scaling and offloading tasks) should be automatically applied.

In this sense, a better support to define complex IoT systems and their (self)adaptation rules to semi-automate the deployment and evolution process is necessary [2]. A usual strategy when it comes to modeling complex domains is to develop a domain-specific language (DSL) for that domain [3]. In short, a DSL offers a set of abstractions and vocabulary closer to the one already employed by domain experts.

Nevertheless, current DSLs for IoT do not typically cover multi-layered architectures [4]–[7] and even less include a sub-language to ease the definition of the dynamic rules governing the IoT system.

Our work aims to cover this gap by proposing a new domain-specific language (DSL) for IoT systems focusing on three main contributions: (1) modeling primitives covering the four layers of an IoT system, including IoT devices (sensors or actuators), edge, fog, and cloud nodes; (2) modeling the deployment and grouping of container-based applications on those nodes; and (3) a specific sublanguage to express adaptation rules to guarantee QoS at runtime. We have implemented the DSL using the Meta Programming System (MPS)[1]. A proof of concept of a generator for deploying the modeled IoT system on a K3S-based (Kubernetes distribution built for IoT and edge computing) infrastructure is also provided.

The remainder of the paper is organized as follows: Section II presents background information. Section III presents the DSL design. Section IV summarizes the related work to this research. In Section V, we present the DSL implementation and code generation. Finally, Section VI summarizes the conclusions and future work.

II. Background and Running Example

Multi-layered architectures have emerged to increase the flexibility of pure cloud deployments and help meet non-functional IoT system requirements. In particular, edge computing and fog computing are solutions that propose to bring computation, storage, communication, control, and decision making closer to IoT devices [8]. Edge and fog computing often leverage containerization (the encapsulation of software

This work has been partially funded by the Spanish government (LOCOSS project - PID2020-114615RB-I00), the Colombian government (*Becas del Bicentenario* program - Art. 45, law 1942 of 2018), and the ECSEL Joint Undertaking (JU) under grant agreement No 101007260.

[1]https://www.jetbrains.com/mps/

code [9]) as a virtualization technology. There are strong similarities between these two solutions, but the main difference is where the processing takes place. While edge computing takes place on devices directly connected to sensors and actuators, or on gateways physically close to these, fog computing takes place on LAN-connected nodes usually farther away from the sensors and actuators. Sensors and actuators compose the device layer. Sensors generate information by monitoring physical variables (such as temperature, motion, and oxygen) and actuators (such as valves, fans, and alarms) are devices capable of transforming energy into activation of a process.

We will use a coal mining industry setting as a running example to better illustrate these concepts. In recent years, the coal mining industry has been significantly improving aspects such as occupational safety, production efficiency, and environmental pollution by implementing IoT and cyber-physical systems to monitor, control, and automate processes [10]. For example, in Colombia, the underground coal mining industry has adopted the use of monitoring systems to alert workers to the presence of toxic and explosives gases inside the underground mines [11]. Although mines commonly consist of several levels, pitheads, tunnels, and working fronts, for the sake of the example, we will analyze a monitoring system for a small mine with one level, two main tunnels, and two work fronts (see Fig. 1). The infrastructure and software features of the gas monitoring system are as follows.

Device layer. Each work front has a temperature sensor, a methane gas sensor, and a visual alarm to alert workers to the existence of gases. Additionally, there is a fan at the entrance of tunnel B to control the temperature and avoid gas concentrations in the mine.

Edge layer. An edge node deployed at each work front receives the information collected by the sensors. These nodes run the App1 containerized application which analyzes sensor data in real time and triggers alarms when methane gas is detected in the environment.

Fog layer. The mine has a fog node that communicates with the Edge nodes. This fog node located at the entrance of tunnel A, runs a containerized application to control the fan speed based on aggregated temperature and methane gas data (App2).

Cloud layer. The cloud layer has a server or cloud node that runs two containers: (DB1) a database to store part of the sensor data, and (App3) a web application to visualize historical information from sensor data and incidents at any of the mine's work fronts.

During system operation, QoS must be guaranteed mainly for critical applications. For example, (App1) real-time analysis to detect and alarm the presence of gas methane should always be available. However, some environmental factors could generate unexpected events that affect system operation. In these cases, the system must self-adapt to guarantee its operation. For instance, if the edge node fails, it is necessary to migrate container App1 to another suitable node. The next section shows how we are able to model all these concepts.

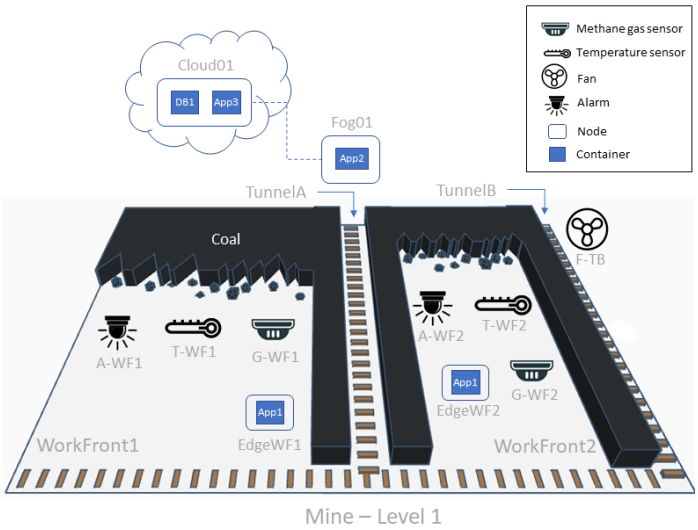


Fig. 1. Overview of the gas monitoring system

III. DSL FOR THE SPECIFICATION OF IOT SYSTEMS

A DSL is defined by three core ingredients [3]: the abstract syntax (i.e., the concepts of the DSL and their relationships), the concrete syntax (the notation to represent those concepts) and its semantics which most of the time is not formalized.

In this section, we present the DSL for modeling the static and dynamic aspects of the IoT system as follows. First, Section III-A describes the abstract syntax and the concrete syntax of the DSL elements for the specification of static aspects including architecture and deployment of containerized applications. Then, Section III-B covers the dynamic ones, i.e. the self-adaptation rules.

A. Modeling of the IoT architecture

1) Abstract syntax: The abstract syntax of the DSL is commonly defined through a metamodel that represents the domain concepts and their relationships. Fig. 2 shows the excerpt of the metamodel that abstracts the concepts to define multi-layer IoT architectures. The sensors and actuators of the device layer are modeled using the *Sensor* and *Actuator* concepts that inherit from the *IoTDevice* concept. This generalization is restricted to be complete and disjoint, each instance of *IoTDevice* must be either a *Sensor* or an *Actuator* but not both. All *IoTDevice*s have a connectivity type (such as ethernet, wifi, ZigBee, or another) represented as an attribute with the *Connectivity* enumeration as type. Because the MQTT messaging protocol is becoming the standard for M2M communications [12], we cover the concepts for specifying this type of communication between IoT devices and nodes. Therefore, *IoTDevices* are publishers or subscribers to a topic MQTT broker specified by the attribute *topic*). The gateway of an *IoTDevice* can be modeled through the *gateway* relationship with the *EdgeNode* concept. Via this gateway, the sensor can communicate with other nodes, e.g. the MQTT broker node.

The location of IoTDevices can be specified by means of geographic coordinates (*latitude* and *longitude* attributes).

Both *Sensors* and *Actuators* have a type represented by the concepts *SensorType* and *ActuatorType*. For instance, following the example scenario (Fig. 1), there are temperature and gas type sensors, and there are fan and alarm type actuators.

Physical (or even virtual) spaces such as tunnels, work fronts, buildings, or mines can be represented by the concept *Region*. A *Region* can contain more regions (relationship *subregion*s in the metamodel). For example, region *Level1* (Fig. 1) contains subregions *WorkFront1*, *WorkFront2*, *TunnelA*, and *TunnelB*. *IoTDevice*s, *EdgeNode*s, and *FogNode*s can be part of regions or subregions. Therefore, the *EdgeWF1* node is located in the *WorkFront1* region of *Level1* of *Mine*, while the *Fog01* node is located in the *TunnelA* region of *Level1*.

Edge, fog and cloud nodes are all instances of *Node*, one of the key concepts of the metamodel. A node is the concept responsible for hosting the software containers. Every node belongs to a single subtype. Communication between nodes can be specified by means of the *linkedNodes* relationship as sometimes we may want to indicate what nodes on a certain layer could act as reference nodes in another layer (e.g. what cloud node should be the first option for a fog node). Nodes can be also grouped in clusters of nodes (typically of the same type but this is not mandatory) that work together. A *Cluster* has a master node (represented by the *master* relationship) and several worker nodes (represented by the *workers* relationship).

A *Node* can host several software containers according to its capabilities and resources. These resources are represented by the attributes *cpuCores*, *memory*, and *storage*. The *containers* composition relationship represents the containers that are hosted and executed on a *Node*. Each software container runs an application (represented by the concept *Application*) that has a minimum required resources specified by the attributes *minRam* and *minStorage*. The repository of the application image is specified through the *imageRepo* attribute.

2) Concrete syntax.: The concrete syntax refers to the type of notation (such as textual, graphical, tabular, or hybrid) to represent the concepts of the metamodel. We take advantage of MPS projectional editors to define a hybrid notation (textual, tabular, and tree view). Projectional editors are editors in which the user's editing actions directly change the Abstract Syntax Tree (AST) without using a parser [13]. That is, while the editing experience simulates that of classical parsing-based editors, there is a single representation of the model stored as an AST and rendered in a variety of perspectives thanks to the corresponding projectional editors that can deal with mixed-language code and support various notations such as tables, mathematical formulas, or diagrams.

Our DSL enables the modeling of all concepts using a textual notation. Additionally, for some concepts we also offer complementary notations that we believe are better suited for that particular concept: a tabular notation for modeling nodes and IoT devices, and a tree notation for regions. These projectional editors are presented in Section V.

3) Example scenario: We present next how to model the IoT architecture of the running example (Section II) using our DSL. Fig. 3(a) shows the textual modeling of application App1 while Fig. 3(b) describes the regions of the example. The *Level1* region has four subregions: two work fronts and two tunnels.

IoT devices can be modeled using a tabular notation. Fig. 4 shows the list of sensors and actuators located in the *WorkFront1* region. For describing the system nodes, we propose a tabular notation as well. The node description includes the layer it belongs to (edge, fog, or cloud), the hardware properties (such as memory and storage resources), the regions where it is located, and the application containers it hosts.

Finally, the concept *Cluster* represents a group of nodes modeled by means of a textual notation in a similar way to the applications. To relate the master node and the worker nodes it is necessary to have previously defined them.

B. Modeling of the Self-Adaptation Rules

The dynamic environment of an IoT system generates unexpected changes that could affect the QoS. The system should be prepared to cope with unexpected changes and self-adapt to different situations. Because of this, our DSL enables the modeling of self-adaptation rules of the system.

1) Abstract syntax: The metamodel excerpt representing the abstract syntax for defining the rules is presented in Fig. 5.

Every rule is an instance of *AdaptationRule* that has a triggering condition which is an expression. We reuse an existing *Expression* language to avoid redefining in our language all the primitive data types and all the operations to manipulate them. Such Expression language could have been for instance the Object Constraint Language (OCL) but to facilitate the implementation of the DSL later on, we directly reused the MPS Baselanguage. The metamodel extends the generic Expression concept by adding sensor and QoS conditions that can be combined also with all other types of expressions (e.g. numerical ones) in a complex conditional expression.

The *QoSEvent* condition is a relational expression that represents a threshold of resource consumption or QoS metrics while a *SensorEvent* represents the occurrence of an event resulting from the analysis of sensor data (e.g., the detection of methane gas in a work front). Note that *SensorEvent* conditions can be linked to specific sensors or to sensor types to express conditions involving a group of sensors.

Similarly, *QoSEvent* type condition allows to check a Metric (such as Latency, CPU consumption, and others) on a specific node or a group of nodes belonging to a *Region* or *Cluster*. For example, the condition $CPU(Mine.edgeNodes) > 50$ is triggered when the CPU consumption on the edge nodes of *Mine* exceeds 50%. The *SensorEvent* type condition allows to check the data of a specific sensor or a group of sensors located in a *Region*. For example, the condition $Mine.Temperature(5) > 40$ is triggered when the temperature reported by at least 5 sensors inside *Mine* exceeds 40.

Moreover, we can define that the condition should be true over a certain period (to avoid firing the rule in reaction to minor disturbances) before executing the rule. Once fired, all or some of the actions are executed in order, depending on the

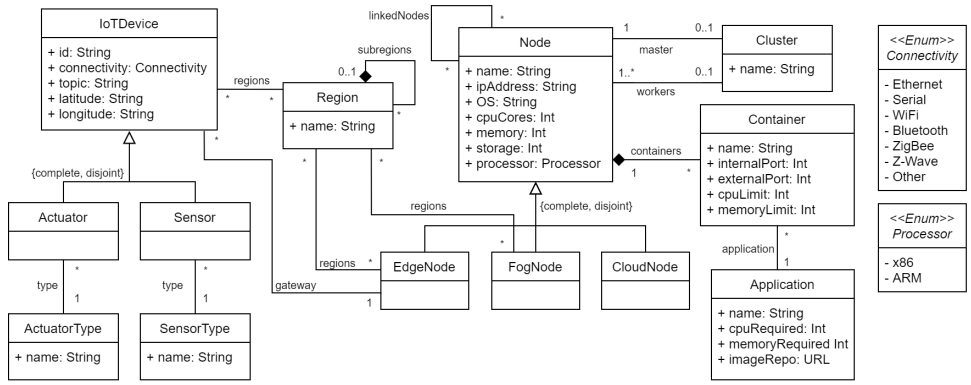
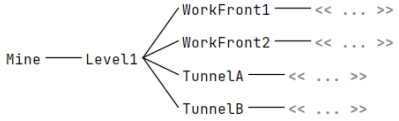


Fig. 2. Excerpt of the metamodel depicting multi-layer architecture and deployment

```
Name: App1
  Memory required: 500 MB
  CPU required: 400 mCores
  Repository: mineIndustry/imageApp1:V0.1
```

(a) Application modeling example



(b) Regions modeling example

Fig. 3. Modeling example

allActions attribute. If set to false, only the number of *Action*s specified by the attribute *actionsQuantity* must be executed starting with the first one in order and continuing until the required number of actions have been successfully applied.

An action can be classified as *Offloading*, *Redeploy*, *Scaling*, or *OperateActuator*. The *Offloading* action consists in migrating a container from a source node to a destination node. This migration can be between nodes of different layers. The *container* relationship represents the container that will be offloaded. The target node is specified by the *targetNode* relationship. However, if the target node does not have the resources to host the container, a cluster or a group of nodes in a *Region* can be specified (*targetRegion* and *targetCluster* relationships) to offload the container. The *Scaling* action involves deploying replicas of an application. This application is represented by the *app* relationship, and the number of replicas to be deployed is defined by the *instances* attribute. The replicas of the application are deployed in one or several nodes of the system represented by the *targetNodes*, *targetCluster*, and *targetRegion* relationships. The *Redeployment* action consists in stopping and redeploying a container running on a node. The container to redeploy is indicated by the *container* relationship. Finally, the *OperateActuator* action is to control the actuators of the system (e.g. to activate or deactivate an alarm). The message attribute represents the message that will be published in the broker and interpreted by the actuator. This action can control actuators that require only one control value such as On/Off.

2) Concrete syntax: The system adaptation rules are specified by a textual notation using as keywords the names of the metaclasses of the abstract syntax. The conditions follow the grammar of a relational expression with the use of mathematical symbols (such as $<$, $>$, and $=$) and logical operators (such as & and $||$). It is worth to note that, given that the textual notation is a projection of the abstract syntax, the MPS editor (see Section V) offers a powerful autocomplete feature to guide the designer through the rule creation process.

3) Example Scenario: As a scenario, we show two adaptation rules for our coal mine example: (1) to guarantee the execution of the *App1* deployed on the *EdgeWF1* node (container *C01*), we modeled the rule as shown in (Fig. 6). This rule offloads the container *C01* hosted on node *EdgeWF1* to a nearby node (e.g., node *EdgeWF2*) when the CPU or RAM consumption exceeds 90% for one minute. If the *EdgeWF2* node does not have the necessary resources to host that new container, a Region (e.g. *Level1*) can be specified so that a suitable node will be searched there. However, if this offloading action cannot be executed, for example, because in *Level1* there is no node capable of hosting the container, then we must define a backup action. Therefore, we have modeled a second action (*Scaling*) to deploy a new container instance of the *App1* application on any of the nodes of the *Mine*. For this adaptation rule only one action (the first or the second one) will be executed. Therefore, the checkbox *Perform all actions* must be unchecked and the number of actions to be performed must be set to one. (2) We model another adaptation rule (see Fig. 7) to activate the alarms of the mine (A-WF1 and A-WF2) when one of the methane sensors (G-WF1 and G-WF2) detects a gas concentration greater than 1.5 for 5 seconds. The ”On” message is published in the broker topics consumed by each actuator (alarms).

IV. Related Work

Modeling of cloud architectures has been widely studied [14]–[16], including provisioning of resources for cloud applications. Nevertheless, these proposals do not cover multi-layered architectures involving fog and edge nodes. This is also the case for Infrastructure-as-code (IaC) tools such as

	Device	ID	Type	Regions	Brand	Communication	Gateway	Topic	Latitude	Longitude
1	Sensor	G-WF1	Smoke	WorkFront1	Microship	ZigBee	EdgeWF1	level1/front1/methane	51° 30′ 30′′ N	0° 7′ 32′′ O
2	Sensor	T-WF1	Temperature	WorkFront1	Cisco	ZigBee	EdgeWF1	level1/front1/temp	51° 30′ 31′′ N	0° 7′ 33′′ O
3	Actuator	A-WF1	Alarm	WorkFront1	Burkert	Z_Wave	EdgeWF1	level1/front1/alarm	51° 30′ 28′′ N	0° 7′ 15′′ O

Fig. 4. IoT devices modeling example (tabular notation)

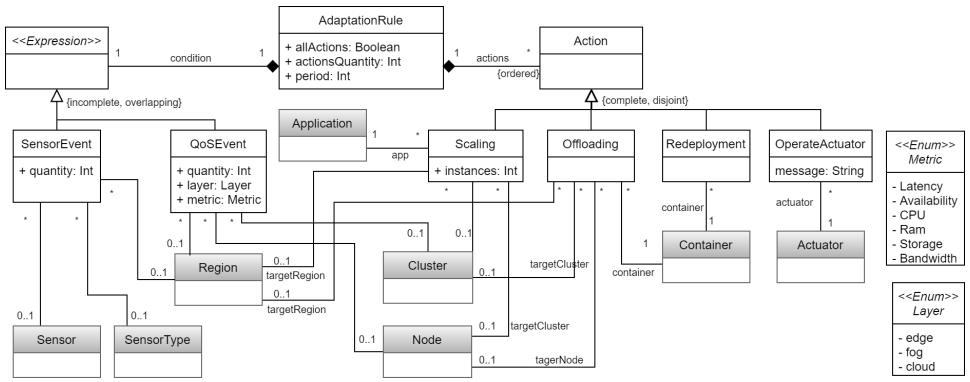


Fig. 5. Excerpt of the metamodel depicting adaptation rules

```
Condition: CPU(EdgeWF1) > 90 || Ram(EdgeWF1) > 90
Period: 60000 ms
Actions
 Perform all actions / Number of actions to be performed: 1
  * Offloading -> Container: C01
                  Target node(s): EdgeWF2
                  Target region(s): Level1
                  Target cluster(s): << ... >>
  * Scaling -> Application: App1
               Instances: 1
               Target node(s): << ... >>
               Target region(s): Mine
               Target cluster(s): << ... >>
```

Fig. 6. Adaptation rule modeling example 1

```
Condition: G-WF1 > 1.5 || G-WF2 > 1.5
Period: 5000 ms
Actions
 Perform all actions
  * Operate Actuator -> Actuator: A-WF1
                        message: On
  * Operate Actuator -> Actuator: A-WF2
                        message: On
```

Fig. 7. Adaptation rule modeling example 2

Terraform, Chef, Ansible, or Puppet and other IoT-focused proposals [5].

A few works such as [6], [17], [18] propose solutions for deploying containerized applications on edge and fog nodes using orchestrators such as Kubernetes and Docker Swarm. However, the deployment process is manual and fixed, with no adaptation policies.

Regarding adaptation policies for IoT or cloud architectures, works such as [19]–[23] propose partial solutions as they either restrict the parts of the system that could be adapted (e.g. only the global monitoring component to adapt to manual changes in the deployed components) or offer some type of adaptation rules but with limited expressiveness and rules that must be manually triggered instead of being self-adaptive.

Works more closely related to our proposal follow. Lee et al. [4] present a self-adaptive framework to dynamically satisfy functional requirements for IoT systems. This framework, based on MAPE loop, enables modeling of the IoT environment through a finite-state machine. However, adaptations are only enabled at the device layer level (e.g., opening a window when the temperature exceeds a limit). Rules for other layers or involving several layers are not supported. Garlan et al. [24] present Rainbow, a framework for adapting a software system when a constraint is violated. The constraints are expressed in terms of performance, cost, or reliability, and the adaptations depend on a set of style specific actions that the control infrastructure can perform. Similarly, Weyns et al. [25] propose MARTA, an architecture-based adaptation approach to automate the management of IoT systems employing runtime models and leveraging the MAPE loop. However, Rainbow and MARTA focus only on architectural adaptations (without addressing container-based applications) and do not cover the specification of multi-layer architectures. Petrovic et al. [9] propose SMADA-Fog, a semantic model-driven approach to deployment and adaptation of container-based applications in Fog computing scenarios. However, SMADA-Fog does not allow the specification of complex adaptation rules composed of various conditions and actions. Moreover, grouping nodes and IoT devices according to their location is not possible, forbidding the possibility to apply adaptations on group of nodes belonging to a cluster or a given region. In [26] we present an in-depth analysis of the state of the art.

To sum up, ours is the first proposal that enables the modeling of multi-layer IoT architectures (device, edge, fog, and cloud) and the definition of complex rules covering all layers (and combinations of) and involving multiple conditions and actions that can, potentially, involve groups of nodes in the same region or cluster of the IoT system.

V. Tool Support

Our DSL is implemented using MPS, an open-source language workbench developed by Jetbrains. In addition, we have developed a proof-of-concept of a code generator. Both DSL and generator are freely available in our repository[2].

Section III describes the abstract syntax and editors specified in MPS. Furthermore, we have used the MPS constraint mechanism to embed several well-formedness rules in the editors. For instance, we have added a constraint to avoid repeated names, a constraint to limit the potential values of some numerical attributes, a constraint to restrict the potential relationships between nodes, etc.

While the DSL formalizes the knowledge about the business in the domain, the code generator encapsulates the implementation of that knowledge in a particular target technology. More specifically, as a proof of concept, we have implemented a model-to-text transformation that generates a YAML handle code to deploy container-based applications with the K3S orchestrator from our IoT system definition.

Due to space limitations, we do not show here the transformation nor the generated code. You can find all this information in the project repository

VI. Conclusion and Future Work

We have presented a DSL for modeling multi-layered architectures of IoT systems and their self-adaptation rules. The DSL is implemented as a projectional editor created with the Jetbrains MPS tool. This gives us the flexibility to offer, and mix, a variety of concrete notations for the different concepts of the DSL. We have also presented a prototype Kubernetes manifest generator for deploying container-based applications.

As further work, we will extend the DSL with primitives to express more complex adaptation strategies and patterns (e.g., canary deployment and rolling update). Additionally, we will study the integration of our DSL with other languages such as SysML. At the tool level, we will work on a visual renderer of the modeled architecture to complement the current projections and complete the code generator to cover the dynamic aspects of the IoT design, potentially by extending a policy engine with adhoc customizations to cover our multi-layered representation. Finally, we plan to validate the DSL in the mining industry (sector providing the funding for this research as well) where IoT monitoring and control systems are heavily used to improve work safety.

References

[1] Y. Jiang, Z. Huang, and D. H. Tsang, "Challenges and solutions in fog computing orchestration," *IEEE Network*, vol. 32, pp. 122–129, 2017.

[2] A. Rhayem, M. B. A. Mhiri, and F. Gargouri, "Semantic web technologies for the internet of things: Systematic literature review," *Internet of Things*, p. 100206, 2020.

[3] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[4] E. Lee, Y.-D. Seo, and Y.-G. Kim, "Self-adaptive framework based on mape loop for internet of things," *sensors*, vol. 19, no. 13, p. 2996, 2019.

[2]https://github.com/SOM-Research/selfadaptive-IoT-DSL.git

[5] P. Patel and D. Cassou, "Enabling high-level application development for the internet of things," *Journal of Systems and Software*, vol. 103, pp. 62–84, 2015.

[6] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: a framework for continuous automated iot application deployment in fog computing," in *IEEE Int. Conference on AI & Mobile Services*, 2017, pp. 38–45.

[7] F. Ciccozzi and R. Spalazzese, "Mde4iot: supporting the internet of things with model-driven engineering," in *International Symposium on Intelligent and Distributed Computing*. Springer, 2016, pp. 67–76.

[8] O. C. A. W. Group *et al.*, "Openfog reference architecture for fog computing," *OPFRA001*, vol. 20817, p. 162, 2017.

[9] N. Petrovic and M. Tosic, "Smada-fog: Semantic model driven approach to deployment and adaptivity in fog computing," *Simulation Modelling Practice and Theory*, vol. 101, p. 102033, 2020.

[10] G. Wang, Y. Xu, and H. Ren, "Intelligent and ecological coal mining as well as clean utilization technology in china: Review and prospects," *International Journal of Mining Science and Technology*, vol. 29, no. 2, pp. 161–169, 2019.

[11] I. Alfonso, C. Goméz, K. Garcés, and J. Chavarriaga, "Lifetime optimization of wireless sensor networks for gas monitoring in underground coal mining," in *2018 7th International Conference on Computers Communications and Control (ICCCC)*. IEEE, 2018, pp. 224–230.

[12] M. Amoretti, R. Pecori, Y. Protskaya, L. Veltri, and F. Zanichelli, "A scalable and secure publish/subscribe-based framework for industrial iot," *IEEE Transactions on Industrial Informatics*, 2020.

[13] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *Proc. of the 24th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, 2016, pp. 763–774.

[14] J. Sandobalin, E. Insfran, and S. Abrahão, "Argon: A model-driven infrastructure provisioning tool," in *ACM/IEEE 22nd Int. Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019, pp. 738–742.

[15] K. Sledziewski, B. Bordbar, and R. Anane, "A dsl-based approach to software development and deployment on cloud," in *24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010, pp. 414–421.

[16] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann, "From architecture modeling to application provisioning for the cloud by combining uml and tosca." in *CLOSER (2)*, 2016, pp. 97–108.

[17] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A containerized big data streaming architecture for edge cloud computing on clustered single-board devices." in *Closer*, 2019, pp. 68–80.

[18] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila, "Docker enabled virtualized nanoservices for local iot edge networks," in *IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2019, pp. 1–7.

[19] L. Cianciaruso, F. di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, and A. Solberg, "Using models at runtime to support adaptable monitoring of multi-clouds applications," in *16th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2014, pp. 401–408.

[20] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "Cloudmf: Model-driven management of multi-cloud applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 18, no. 2, pp. 1–24, 2018.

[21] W. Chen, C. Liang, Y. Wan, C. Gao, G. Wu, J. Wei, and T. Huang, "More: A model-driven operation service for cloud-based it systems," in *IEEE Int. Conference on Services Computing*, 2016, pp. 633–640.

[22] T. Holmes, "Facilitating migration of cloud infrastructure services: A model-based approach." in *CloudMDE@ MoDELS*, 2015, pp. 7–12.

[23] J. Erbel, F. Korte, and J. Grabowski, "Comparison and runtime adaptation of cloud application topologies based on occi." in *CLOSER*, 2018, pp. 517–525.

[24] D. Garlan, B. Schmerl, and S.-W. Cheng, "Software architecture-based self-adaptation," in *Autonomic computing and networking*. Springer, 2009, pp. 31–55.

[25] D. Weyns, M. U. Iftikhar, D. Hughes, and N. Matthys, "Applying architecture-based adaptation to automate the management of internet-of-things," in *European Conference on Software Architecture*. Springer, 2018, pp. 49–67.

[26] I. Alfonso, K. Garcés, H. Castro, and J. Cabot, "Self-adaptive architectures in iot systems: A systematic literature review," 2021, arXiv:2109.03312.

APPENDIX

Each metaclass in our DSL is defined as a *concept* in in MPS. For each concept it is necessary to define several items such as name, properties, and relations. For example, the definition of the *IoTDevice* concept is presented in Fig. 8. This abstract concept has a set of attributes defined as properties, an association relation (*gateway*) with multiplicity of 1, and a smart reference (*regions*). Smart references are the way to define associations with 0..n cardinalities in MPS.

The definition of all the metamodel elements as MPS concepts is available in our public repository.

```
abstract concept IoT_Device extends    BaseConcept
                           implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  brand         : string
  communication : Connectivity
  topic         : string
  latitude      : string
  longitude     : string

  children:
  regions : Region_Reference[0..n]

  references:
  gateway   : Edge_Node[1]
```

Fig. 8. *IoTDevice* concept definition in MPS

The MPS TextGen module supports the definition of model-to-text transformations. We have used it to implement our DSL-to-K3S generator.

More specifically, in our mapping strategy, each modeled container is deployed into a pod. For this, we generate the code of a deployment artifact and a service artifact in a yaml file. An excerpt of the TextGen code for the container concept is as shown in Fig. 9.

```
append {---} \n;
append {apiVersion: v1} \n;
append {kind: Service} \n;
append {metadata:} \n;
with indent {
  append indent {name: } ${node.name} {-entrypoint} \n;
  append indent {namespace: default} \n;
}
append {spec:} \n;
with indent {
  append indent {type: NodePort} \n;
  append indent {selector:} \n;
  append indent indent {app: } ${node.name} {-} ${node.application.name} \n;
  append indent {ports:} \n;
  append indent {- port: } ${"" + node.internalPort} \n;
  append indent {  nodePort: } ${"" + node.externalPort} \n \n;
}
append {---} \n;
```

Fig. 9. Excerpt of the TextGen Container concept

Listing 1 shows the code generated for the K3S deployment and service of the software container "C01". A pod is configured with a container with the repository image *mineIndustry/imageApp1:V0.1* specified by the label *image*; by the label *nodeSelector* the node with the tag *EdgeWF1* is assigned to host the pod; finally, the application container is exposed via port 8000 (*externalPort* attribute of the *Container* concept).

Listing 1. Excerpt of generated yaml code

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: C01-deployment
spec:
  replicas: 1
  selector:
    matchlabels:
      app: C01-App1
  template:
    metadata:
      labels:
        app: C01-App1
    spec:
      containers:
      - name: C01
        image: mineIndustry/imageApp1:V0.1
        ports:
          - containerPort: 8080
    nodeSelector:
      node: EdgeWF1
---
apiVersion: v1
kind: Service
metadata:
  name: C01-entrypoint
  namespace: default
spec:
  type: NodePort
  selector:
    app: C01-App1
  ports:
  - port: 8080
    nodePort: 8000
```